Lecture 7: Policy Gradient

David Silver

overall

lec 3:

Planning (prediction, control) by DP.
Solve a known MDP.

Lec 4 ■

Drop your agent in an unknown MDP with a given policy, how to evaluate this policy, how much rewards we can get if following the behaviors of this policy.

- Model-free prediction
- *Estimate* the value function of an *unknown* MDP

Lec 5 ■

- Model-free control
- *Optimise* the value function of an *unknown* MDP

Find v_*, q_*
We use same tools, we iterate them and find the best possible behaviors.

Lec 6 scale up to real practical problems. (Still model free)

Previous lectures: Use table
Scale Up: Use function approximation.

Lec 6: Function approximation for value based algorithms
Lec 7: Function approximation for policy based algorithms

Outline

1 Introduction

Methods that optimize the policy directly (instead of working on value func)
Below 3 methods all base on the idea of policy gradient

2 Finite Difference Policy Gradient

Naive, effective numerical approach

3 Monte-Carlo Policy Gradient

Analytic formulation for the policy gradient. An agent follow the trajectory and adjust to the direction with better policy.
Smooth and slow. High iterations and high variance.

4 Actor-Critic Policy Gradient

Most practical set of methods. Combine everything from this and previous classes. Work with value func and with policies.
Smooth and efficient. Low variance.

Policy-Based Reinforcement Learning

State–value func

- In the last lecture we approximated the value or action-value function using parameters θ,

Parametric the value func:

Parameter functions w, like neural network, to fit the ground truth.

$$V_\theta(s) \approx V^\pi(s)$$

True state–value func. How much rewards I can get if follow pi. Oracle tell us this.

$$Q_\theta(s,a) \approx Q^\pi(s,a)$$

True action–value func. True rewards we get.

But the last lecture, we didn't represent the policy explicitly. Instead we just used value func to pick actions. (Q is enough for us to pick actions)

- A policy was generated directly from the value function
  - e.g. using ϵ-greedy
- In this lecture we will directly parametrise the policy

Instead of parametric value func in last lecture.

u VS w:
Parameter vector u
Instead of parameter vector w in last lecture.

$$\pi_\theta(s,a) = \mathbb{P}\left[a \mid s, \theta\right]$$

- We will focus again on model-free reinforcement learning

This lecture: We parametric the distribution of actions in policy pi. We directly manipulate policy pi. We control parameters which affect the distribution by which we pick actions. For input states, parameter vector u (like neural network) give you output: which action you should take or distributions of actions. Then we will see how to adjust the parameter vector u to solve the RL problem.

Motivation:

We want to scale up.

It’s impossible to go over all possible actions/states to know which is best. It’s impossible for each distinct state to say which action I’m going to take. So we approximate the state-value or action value functions based on our experiences (we don’t know the whole picture of the environment (model-free)).

Next:

We need to understand how we take the parametrized policy with parameter vector u, and start to adjust the parameters to get more rewards.

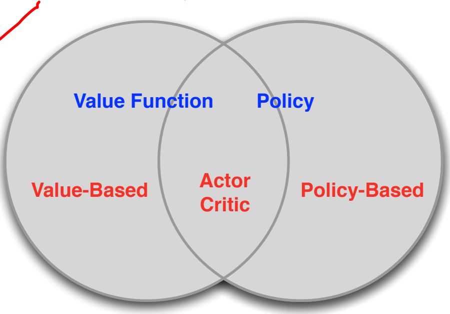
The main mechanism is gradient ascent. How could we compute the gradient to follow to make policy better. If you follow that gradient we will strictly be moving uphill in a way to improve our policy. If we know the gradient wrt the total rewards then we just follow it and will get more rewards.

In next slides, we formalize these and see how we max the rewards. There are few ways we can formalize the objective function.

Value-Based and Policy-Based RL

We don't explicitly represent the policy. We act greedy wrt to Q and sometimes randomly.

- Value Based
    - Learnt Value Function
    - Implicit policy (e.g. ϵ-greedy)
- Policy Based
    - No Value Function
    - Learnt Policy
- Actor-Critic
    - Learnt Value Function
    - Learnt Policy Directly parameterize the policy.



Advantages of Policy-Based RL

With a value based method, you have to solve how to compute the max. Policy gradient methods avoid this max. Like salsa/Q–learning, once we got a Q, all you need to pick actions is maximize over a, and you pick the action with highest q then done. You find the best way to behave. But what if the maximization itself is prohibitively expensive, like a trillion actions or continuous action spaces.

In policy gradient, you incrementally understand what the max will be instead of estimating the max directly every step.

Advantages:

More efficient to represent the policy than the value function. The value func might be super complicated. A policy can be more compact.

- Better convergence properties   More stable. Smoothly update the policy. No dramatic swings, chatter.
- No. 1 reason to use policy gradient: Effective in high-dimensional or continuous action spaces
- Can learn stochastic policies

Disadvantages:

- Typically converge to a local rather than global optimum
- Naive policy based RL: Evaluating a policy is typically inefficient and high variance

Value–based methods use max which is aggressive. But policy gradient take a little step on that direction and smoothly update –> more stable, less efficient. But we can avoid/improve this later (actor–critic).

Example: Rock-Paper-Scissors

if max at every step, then deterministic policy might be good enough? Bc maximizing can be a deterministic process. But sometimes we want stochastic policy.

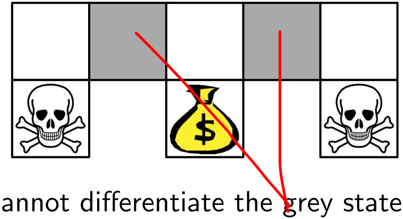
- Two-player game of rock-paper-scissors
  - Scissors beats paper
  - Rock beats scissors
  - Paper beats rock
- Consider policies for *iterated* rock-paper-scissors
  - A deterministic policy is easily exploited
  - Any
  - A uniform random policy is optimal (i.e. Nash equilibrium)

This is an example where optimal policy is stochastic.

Example: Aliased Gridworld (1)

partially observed environment:
(use features)

Full observed MDP has perfect state representation (but here we don't).



- The agent cannot differentiate the grey states
- Consider features of the following form (for all N, E, S, W)

Feature contain both state and action.

$$\phi(s, a) = \mathbf{1}(\text{wall to N}, a = \text{move E})$$

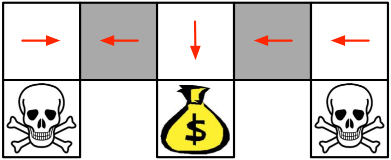
- Compare value-based RL, using an approximate value function

$$Q_\theta(s, a) = f(\phi(s, a), \theta)$$

- To policy-based RL, using a parametrised policy

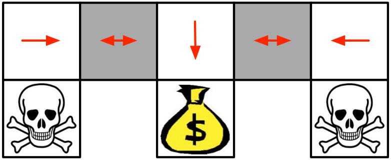
$$\pi_\theta(s, a) = g(\phi(s, a), \theta)$$

Example: Aliased Gridworld (2)



- Under aliasing, an optimal deterministic policy will either
  - move W in both grey states (shown by red arrows)
  - move E in both grey states
- Either way, it can get stuck and *never* reach the money
- Value-based RL learns a near-deterministic policy
  - e.g. greedy or ϵ-greedy
- So it will traverse the corridor for a long time

Example: Aliased Gridworld (3)



- An optimal stochastic policy will randomly move E or W in grey states

$$\pi_\theta(\text{wall to N and S, move E}) = 0.5$$
$$\pi_\theta(\text{wall to N and S, move W}) = 0.5$$

- It will reach the goal state in a few steps with high probability
- Policy-based RL can learn the optimal stochastic policy

Deterministic and value based cannot.

When arising happens, or features you use limit your view of the world (equivalent to partially observed mdp), the optimal can be stochastic.

How to optimize the policy? 1. What is the objective? 2. How to optimize?

Policy Objective Functions

- Goal: given policy $\pi_\theta(s, a)$ with parameters θ, find best θ
- But how do we measure the quality of a policy π_θ?
- In episodic environments we can use the start value

$$J_1(\theta) = V^{\pi_\theta}(s_1) = \mathbb{E}_{\pi_\theta}[v_1]$$

If I always start with state s1, what's the total rewards I will get from state s1 onwards.

- In continuing environments we can use the average value

$$J_{avV}(\theta) = \sum_s \underbrace{d^{\pi_\theta}(s)}_{\text{Distribution}} \underbrace{V^{\pi_\theta}(s)}_{\text{The value from that state onwards}}$$

- Or the average reward per time-step

$$J_{avR}(\theta) = \sum_s d^{\pi_\theta}(s) \sum_a \pi_\theta(s, a) \mathcal{R}_s^a$$

Take the average of my immediate rewards over the entire distribution of states that I visit

Fortunately, policy gradient applies for all these 3 scenarios.

- where $d^{\pi_\theta}(s)$ is stationary distribution of Markov chain for π_θ

Policy Optimisation

- Policy based reinforcement learning is an optimisation problem
- Find θ that maximises $J(\theta)$
- Some approaches do not use gradient

Optimization methods can be classified as:
– gradient based methods
– gradient free methods

If you don't have access to gradient

  - Hill climbing
  - Simplex / amoeba / Nelder Mead
  - Genetic algorithms
- Greater efficiency often possible using gradient

If you have access to gradient

  - Gradient descent
  - Conjugate gradient
  - Quasi-newton
- We focus on gradient descent, many extensions possible

Like second order method, quasi newton.

- And on methods that exploit sequential structure

Policy Gradient

- Let $J(\theta)$ be any policy objective function
- Policy gradient algorithms search for a *local* maximum in $J(\theta)$ by ascending the gradient of the policy, w.r.t. parameters θ

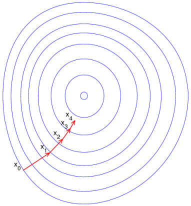
$$\Delta\theta = \alpha \nabla_\theta J(\theta)$$

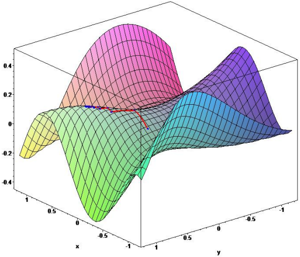
- Where $\nabla_\theta J(\theta)$ is the policy gradient

$$\nabla_\theta J(\theta) = \begin{pmatrix} \frac{\partial J(\theta)}{\partial \theta_1} \\ \vdots \\ \frac{\partial J(\theta)}{\partial \theta_n} \end{pmatrix}$$

The gradient of the objective func is the vector of the partial derivatives

- and α is a step-size parameter





if you cannot compute gradient directly:

Computing Gradients By Finite Differences

- To evaluate policy gradient of $\pi_\theta(s, a)$
- For each dimension $k \in [1, n]$
  - Estimate kth partial derivative of objective function w.r.t. θ
  - By perturbing θ by small amount ϵ in kth dimension

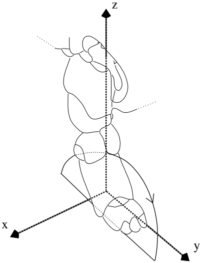
  Get numerical estimate of the gradient:

$$\frac{\partial J(\theta)}{\partial \theta_k} \approx \frac{J(\theta + \epsilon u_k) - J(\theta)}{\epsilon}$$

  If I perturb it a little bit, we look at the difference between the objectives before and after I perturb it. This can be the estimated gradient in this direction.

  where u_k is unit vector with 1 in kth component, 0 elsewhere
- Uses n evaluations to compute policy gradient in n dimensions
- Simple, noisy, inefficient - but sometimes effective
- Works for arbitrary policies, even if policy is not differentiable 可微的

Training AIBO to Walk by Finite Difference Policy Gradient



- Goal: learn a fast AIBO walk (useful for Robocup)
- AIBO walk policy is controlled by 12 numbers (elliptical loci)
- Adapt these parameters by finite difference policy gradient
- Evaluate performance of policy by field traversal time

AIBO Walk Policies

some video playing here

- Before training
- During training
- After training

Simplest approach, no value func yet.

Score Function

- We now compute the policy gradient *analytically*
- Assume policy π_θ is differentiable whenever it is non-zero
- and we know the gradient $\nabla_\theta \pi_\theta(s, a)$
- Likelihood ratios exploit the following identity

$$\begin{aligned}\nabla_\theta \pi_\theta(s, a) &= \overset{==1}{\pi_\theta(s, a) \frac{\nabla_\theta \pi_\theta(s, a)}{\pi_\theta(s, a)}} \\ &= \pi_\theta(s, a) \nabla_\theta \log \pi_\theta(s, a)\end{aligned}$$

- The score function is $\nabla_\theta \log \pi_\theta(s, a)$

This is the thing which tells you how to adjust your policy in a direction that gets more of sth.

Rewriting the gradient in this way, we are able to take expectations.

Next: see what the score function looks like in 2 common examples.

Softmax Policy

Simplest example. Discrete actions.

The softmax policy is basically sth where we want to have some smoothly parameterized policy that tells use how frequently we should choose an action for each of our discrete set of actions. It can be alternative to epsilon–greedy.

What to do:

- We will use a softmax policy as a running example
- Weight actions using linear combination of features $\phi(s,a)^\top\theta$

Consider this as some kind of values to tell us how much we'd like to take that action a.

- Probability of action is proportional to exponentiated weight

Then we turn this into probability.

$$\pi_\theta(s,a) \propto e^{\phi(s,a)^\top\theta}$$

Softmax in general is a policy that is proportional to some exponentiated value. You can choose this value to be anything you want, here we make it to be the “linear combination of features”. This is a good way to parameterize a policy.

- The score function is

Like Atari, we should go left or right? We use some features of going left and right, we weight each of those features. Whichever one scores more highly when we make this weighted sum, would get a higher probability when we actually come to pick actions.

Then we did get the gradient, so we can know how to adjust to get more rewards/ scores. So we need to know the score function:

$$\nabla_\theta \log \pi_\theta(s,a) = \underbrace{\phi(s,a)}_{\text{The feature of the action we actually took}} - \underbrace{\mathbb{E}_{\pi_\theta}\left[\phi(s,\cdot)\right]}_{\text{Average feature for all actions that we might have taken (The usual).}}$$

Totally, if an action gets more than the usual then that's the direction we adjust our policy to do more that action.

Gaussian Policy

Continuous actions. Like AIBO.

We parameterized the mean of the gaussian and you have some randomness around the mean (variance around the mean). So it means most time I will take the mean, sometimes I will take the derivation from that mean (which can be sigma square)

- In continuous action spaces, a Gaussian policy is natural
- Mean is a linear combination of state features $\mu(s) = \phi(s)^\top \theta$
- Variance may be fixed σ^2, or can also parametrised
- Policy is Gaussian, $a \sim \mathcal{N}(\mu(s), \sigma^2)$

  The action a is sleeted according to the normal distribution.
- The score function is

How better than usual when we took this action a.

The action we took The mean action

$$\nabla_\theta \log \pi_\theta(s, a) = \frac{(a - \mu(s))\phi(s)}{\sigma^2}$$

scale

One-Step MDPs

Firstly derive it from the simplest case: one step MDPs. Then extend to multiple–step MDPs.

- Consider a simple class of one-step MDPs
  - Starting in state $s \sim d(s)$
  - Terminating after one time-step with reward $r = \mathcal{R}_{s,a}$

No sequences in this case.

If we want to solve this, we need to find the policy gradient.

- Use likelihood ratios to compute the policy gradient

Objective function:
All 2 types of objective functions we talked about in page12 are same in this case.

$$
\begin{aligned}
J(\theta) &= \mathbb{E}_{\pi_\theta}\left[r\right] \\
&= \sum_{s \in \mathcal{S}} d(s) \sum_{a \in \mathcal{A}} \pi_\theta(s,a)\mathcal{R}_{s,a}
\end{aligned}
$$

The gradient of the whole thing:

$$
\begin{aligned}
\underline{\nabla_\theta J(\theta)} &= \sum_{s \in \mathcal{S}} d(s) \sum_{a \in \mathcal{A}} \pi_\theta(s,a)\nabla_\theta \log \pi_\theta(s,a)\mathcal{R}_{s,a} \\
&= \mathbb{E}_{\pi_\theta}\left[\nabla_\theta \log \pi_\theta(s,a) r\right]
\end{aligned}
$$

Expend using our likelihood ratio trick.
Then the gradient of the whole thing becomes the policy * the gradient of the log policy * the reward you get. This is also an expectation.

This tells us how to adjust our policy to get more or less rewards.

Policy Gradient Theorem

- The policy gradient theorem generalises the likelihood ratio approach to multi-step MDPs
- Replaces instantaneous reward r with long-term value $Q^{\pi}(s, a)$
- Policy gradient theorem applies to start state objective, average reward and average value objective

Theorem

For any differentiable policy $\pi_\theta(s, a)$,
for any of the policy objective functions $J = J_1, J_{avR}$, or $\frac{1}{1-\gamma} J_{avV}$,
the policy gradient is

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}\left[\nabla_\theta \log \pi_\theta(s, a)\ Q^{\pi_\theta}(s, a)\right]$$

Replace immediate reward in one-step MDPs

Monte-Carlo Policy Gradient (REINFORCE)

Use Policy Gradient Theorem to make an algorithm:

- Update parameters by stochastic gradient ascent
- Using policy gradient theorem
- Using return v_t as an unbiased sample of $Q^{\pi_\theta}(s_t, a_t)$

$$\Delta\theta_t = \alpha \nabla_\theta \log \pi_\theta(s_t, a_t) v_t$$

This is forward view so you run it until the end of the episode and you store the rewards for each step. Then you go over each steps again, adjust.

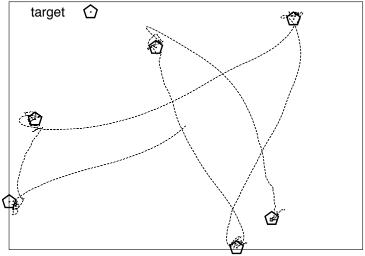
```
function REINFORCE
    Initialise $\theta$ arbitrarily
    for each episode $\{s_1, a_1, r_2, ..., s_{T-1}, a_{T-1}, r_T\} \sim \pi_\theta$ do
        for $t = 1$ to $T - 1$ do
            $\theta \leftarrow \theta + \alpha \nabla_\theta \log \pi_\theta(s_t, a_t) v_t$
        end for
    end for
    return $\theta$
end function
```

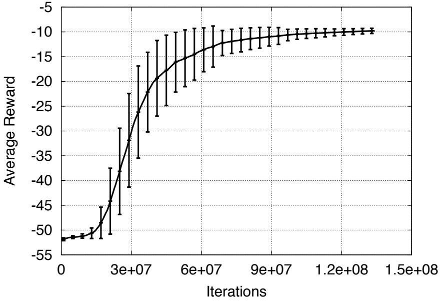
Each step we adjust our parameters in the direction of this score * the return we actually got from that point onwards

Take away:
1. Smooth learning curve. Value based reinforcement learning tends to be much more jaggy.
2. The scale here. 100 million iterations here to solve the problem. So it's slow and very high variance.

The rest of the class: use similar idea but make it more efficient. Reduce the variance.

Puck World Example





- Continuous actions exert small force on puck
- Puck is rewarded for getting close to target
- Target location is reset every 30 seconds
- Policy is trained using variant of Monte-Carlo policy gradient

Reducing Variance Using a Critic

We use value func as estimation of the gradient.

- Monte-Carlo policy gradient still has high variance How to reduce:
- We use a critic to estimate the action-value function,

$$Q_w(s,a) \approx Q^{\pi_\theta}(s,a) \leftarrow r$$

Last lecture: value function approximation. Here we use it again in policy gradient.

- Actor-critic algorithms maintain *two* sets of parameters

  Critic Updates action-value function parameters w

  Actor Updates policy parameters θ, in direction suggested by critic

- Actor-critic algorithms follow an *approximate* policy gradient

$$\nabla_\theta J(\theta) \approx \mathbb{E}_{\pi_\theta}\left[\nabla_\theta \log \pi_\theta(s,a)\ Q_w(s,a)\right]$$
$$\Delta\theta = \alpha \nabla_\theta \log \pi_\theta(s,a)\ Q_w(s,a)$$

replace Q^pi

Estimating the Action-Value Function

Last 2 lectures

- The critic is solving a familiar problem: policy evaluation
- How good is policy π_θ for current parameters θ?
- This problem was explored in previous two lectures, e.g.
  - Monte-Carlo policy evaluation
  - Temporal-Difference learning
  - TD(λ)
- Could also use e.g. least-squares policy evaluation

Action-Value Actor-Critic

- Simple actor-critic algorithm based on action-value critic
- Using linear value fn approx. $Q_w(s, a) = \phi(s, a)^\top w$

Features of state and action * critical weights

  Critic Updates w by linear TD(0)

  Actor Updates θ by policy gradient

```
function QAC
    Initialise $s$, $\theta$
    Sample $a \sim \pi_\theta$
    for each step do
        Sample reward $r = \mathcal{R}_s^a$; sample transition $s' \sim \mathcal{P}_{s,\cdot}^a$
        Sample action $a' \sim \pi_\theta(s', a')$
        $\delta = r + \gamma Q_w(s', a') - Q_w(s, a)$
        $\theta = \theta + \alpha \nabla_\theta \log \pi_\theta(s, a) Q_w(s, a)$
        $w \leftarrow w + \beta \delta \phi(s, a)$
        $a \leftarrow a', s \leftarrow s'$
    end for
end function
```

Q Actor Critic: online algorithm: Every step of the algorithm, we don't need to wait until the end of the episode (like MC). Every single step we perform an update using TD as critic.

Pick an action according to our policy.

Get TD error between the value before that step and the value after that step. Update the critic in the direction of TD error * features. In a direction that minimizes the error between what we thought was happening before what we thought the value was and what the value ended up being after one step using a canonical case using linear TD.

This can be thought as another form of generalized policy iteration. We start with a policy (can be arbitrary: what ever parameters you want), we evaluate that policy using critic. Then instead of using greedy policy improvement, we're moving a gradient step in the direction to get a better policy.

Q: If we use policy gradient methods, do we still have the guarantee that we will find an unique global optimum or could we get tracked in some local optimum?

A:
In the case of table lookup representations of policy, you represent you value function by having one value for each state. If you use bellman equation you get a contraction you guarantee that you get a global optimum.

We policy based methods, if you just follow the gradient, for example the softmax, it's known that for softmax that you also find the global optimum in the table look up case. So if you have a softmax which is like a separate softmax parameters for each state you also achieve a global optimum.

For the case where you get more general function approximator, it's clear that if you get something like neural network neither method will guarantee that you find a global optimum, you can always get trapped in some local optimum.

For certain special cases in between, it's unclear, it's an open research problem.

Adding more tricks to reduce variance and increase efficiency

Bias in Actor-Critic Algorithms

- Approximating the policy gradient introduces bias
- A biased policy gradient may not find the right solution
    - e.g. if $Q_w(s, a)$ uses aliased features, can we solve gridworld example?
- Luckily, if we choose value function approximation carefully
- Then we can avoid introducing any bias
- i.e. We can still follow the *exact* policy gradient

Compatible Function Approximation

Theorem (Compatible Function Approximation Theorem)

If the following two conditions are satisfied:

1. *Value function approximator is compatible to the policy*

$$\nabla_w Q_w(s,a) = \nabla_\theta \log \pi_\theta(s,a)$$

2. *Value function parameters w minimise the mean-squared error*

$$\varepsilon = \mathbb{E}_{\pi_\theta}\left[\left(Q^{\pi_\theta}(s,a) - Q_w(s,a)\right)^2\right]$$

Then the policy gradient is exact,

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}\left[\nabla_\theta \log \pi_\theta(s,a)\ Q_w(s,a)\right]$$

Proof of Compatible Function Approximation Theorem

If w is chosen to minimise mean-squared error, gradient of ε w.r.t. w must be zero,

$$\begin{aligned}\nabla_w \varepsilon &= 0 \\ \mathbb{E}_{\pi_\theta}\left[(Q^\theta(s,a) - Q_w(s,a))\nabla_w Q_w(s,a)\right] &= 0 \\ \mathbb{E}_{\pi_\theta}\left[(Q^\theta(s,a) - Q_w(s,a))\nabla_\theta \log \pi_\theta(s,a)\right] &= 0 \\ \mathbb{E}_{\pi_\theta}\left[Q^\theta(s,a)\nabla_\theta \log \pi_\theta(s,a)\right] &= \mathbb{E}_{\pi_\theta}\left[Q_w(s,a)\nabla_\theta \log \pi_\theta(s,a)\right]\end{aligned}$$

So $Q_w(s,a)$ can be substituted directly into the policy gradient,

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}\left[\nabla_\theta \log \pi_\theta(s,a) Q_w(s,a)\right]$$

Reducing Variance Using a Baseline

- We subtract a baseline function $B(s)$ from the policy gradient
- This can reduce variance, without changing expectation

$$\begin{aligned}\mathbb{E}_{\pi_\theta}\left[\nabla_\theta \log \pi_\theta(s,a) B(s)\right] &= \sum_{s \in \mathcal{S}} d^{\pi_\theta}(s) \sum_a \nabla_\theta \pi_\theta(s,a) B(s) \\ &= \sum_{s \in \mathcal{S}} d^{\pi_\theta} B(s) \nabla_\theta \sum_{a \in \mathcal{A}} \pi_\theta(s,a) \\ &= 0\end{aligned}$$

- A good baseline is the state value function $B(s) = V^{\pi_\theta}(s)$
- So we can rewrite the policy gradient using the advantage function $A^{\pi_\theta}(s,a)$

$$\begin{aligned}A^{\pi_\theta}(s,a) &= Q^{\pi_\theta}(s,a) - V^{\pi_\theta}(s) \\ \nabla_\theta J(\theta) &= \mathbb{E}_{\pi_\theta}\left[\nabla_\theta \log \pi_\theta(s,a)\ A^{\pi_\theta}(s,a)\right]\end{aligned}$$

Estimating the Advantage Function (1)

- The advantage function can significantly reduce variance of policy gradient
- So the critic should really estimate the advantage function
- For example, by estimating *both* $V^{\pi_\theta}(s)$ *and* $Q^{\pi_\theta}(s,a)$
- Using two function approximators and two parameter vectors,

$$\begin{aligned} V_v(s) &\approx V^{\pi_\theta}(s) \\ Q_w(s,a) &\approx Q^{\pi_\theta}(s,a) \\ A(s,a) &= Q_w(s,a) - V_v(s) \end{aligned}$$

- And updating *both* value functions by e.g. TD learning

Estimating the Advantage Function (2)

- For the true value function $V^{\pi_\theta}(s)$, the TD error δ^{π_θ}

$$\delta^{\pi_\theta} = r + \gamma V^{\pi_\theta}(s') - V^{\pi_\theta}(s)$$

- is an unbiased estimate of the advantage function

$$\begin{aligned}\mathbb{E}_{\pi_\theta}\left[\delta^{\pi_\theta} | s, a\right] &= \mathbb{E}_{\pi_\theta}\left[r + \gamma V^{\pi_\theta}(s') | s, a\right] - V^{\pi_\theta}(s) \\ &= Q^{\pi_\theta}(s, a) - V^{\pi_\theta}(s) \\ &= A^{\pi_\theta}(s, a)\end{aligned}$$

- So we can use the TD error to compute the policy gradient

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}\left[\nabla_\theta \log \pi_\theta(s, a) \; \delta^{\pi_\theta}\right]$$

- In practice we can use an approximate TD error

$$\delta_v = r + \gamma V_v(s') - V_v(s)$$

- This approach only requires one set of critic parameters v

Critics at Different Time-Scales

- Critic can estimate value function $V_\theta(s)$ from many targets at different time-scales From last lecture...
  - For MC, the target is the return v_t

$$\Delta\theta = \alpha(v_t - V_\theta(s))\phi(s)$$

  - For TD(0), the target is the TD target $r + \gamma V(s')$

$$\Delta\theta = \alpha(r + \gamma V(s') - V_\theta(s))\phi(s)$$

  - For forward-view TD(λ), the target is the λ-return v_t^λ

$$\Delta\theta = \alpha(v_t^\lambda - V_\theta(s))\phi(s)$$

  - For backward-view TD(λ), we use eligibility traces

$$\begin{aligned}
\delta_t &= r_{t+1} + \gamma V(s_{t+1}) - V(s_t) \\
e_t &= \gamma\lambda e_{t-1} + \phi(s_t) \\
\Delta\theta &= \alpha\delta_t e_t
\end{aligned}$$

Actors at Different Time-Scales

- The policy gradient can also be estimated at many time-scales

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}\left[\nabla_\theta \log \pi_\theta(s,a)\; A^{\pi_\theta}(s,a)\right]$$

- Monte-Carlo policy gradient uses error from complete return

$$\Delta\theta = \alpha(v_t - V_v(s_t))\nabla_\theta \log \pi_\theta(s_t, a_t)$$

- Actor-critic policy gradient uses the one-step TD error

$$\Delta\theta = \alpha(r + \gamma V_v(s_{t+1}) - V_v(s_t))\nabla_\theta \log \pi_\theta(s_t, a_t)$$

Policy Gradient with Eligibility Traces

- Just like forward-view TD(λ), we can mix over time-scales

$$\Delta\theta = \alpha(v_t^\lambda - V_v(s_t))\nabla_\theta \log \pi_\theta(s_t, a_t)$$

- where $v_t^\lambda - V_v(s_t)$ is a biased estimate of advantage fn
- Like backward-view TD(λ), we can also use eligibility traces
  - By equivalence with TD(λ), substituting $\phi(s) = \nabla_\theta \log \pi_\theta(s, a)$

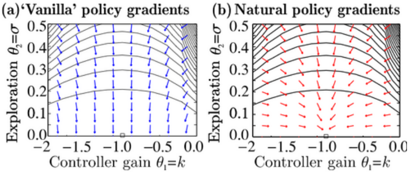
$$\begin{aligned}\delta &= r_{t+1} + \gamma V_v(s_{t+1}) - V_v(s_t) \\ e_{t+1} &= \lambda e_t + \nabla_\theta \log \pi_\theta(s, a) \\ \Delta\theta &= \alpha\delta e_t\end{aligned}$$

- This update can be applied online, to incomplete sequences

Alternative Policy Gradient Directions

- Gradient ascent algorithms can follow *any* ascent direction
- A good ascent direction can significantly speed convergence
- Also, a policy can often be reparametrised without changing action probabilities
- For example, increasing score of all actions in a softmax policy
- The vanilla gradient is sensitive to these reparametrisations

Natural Policy Gradient



- The natural policy gradient is parametrisation independent
- It finds ascent direction that is closest to vanilla gradient, when changing policy by a small, fixed amount

$$\nabla_\theta^{nat} \pi_\theta(s,a) = G_\theta^{-1} \nabla_\theta \pi_\theta(s,a)$$

- where G_θ is the Fisher information matrix

$$G_\theta = \mathbb{E}_{\pi_\theta}\left[\nabla_\theta \log \pi_\theta(s,a) \nabla_\theta \log \pi_\theta(s,a)^T\right]$$

Natural Actor-Critic

- Using compatible function approximation,

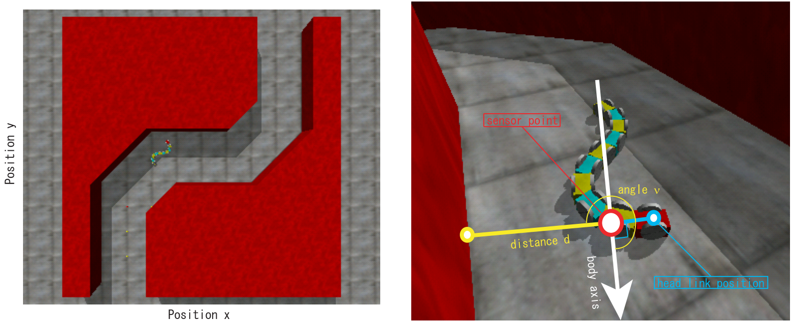
$$\nabla_w A_w(s,a) = \nabla_\theta \log \pi_\theta(s,a)$$

- So the natural policy gradient simplifies,

$$\begin{aligned}\nabla_\theta J(\theta) &= \mathbb{E}_{\pi_\theta}\left[\nabla_\theta \log \pi_\theta(s,a) A^{\pi_\theta}(s,a)\right] \\ &= \mathbb{E}_{\pi_\theta}\left[\nabla_\theta \log \pi_\theta(s,a) \nabla_\theta \log \pi_\theta(s,a)^T w\right] \\ &= G_\theta w \\ \nabla_\theta^{nat} J(\theta) &= w\end{aligned}$$

- i.e. update actor parameters in direction of critic parameters

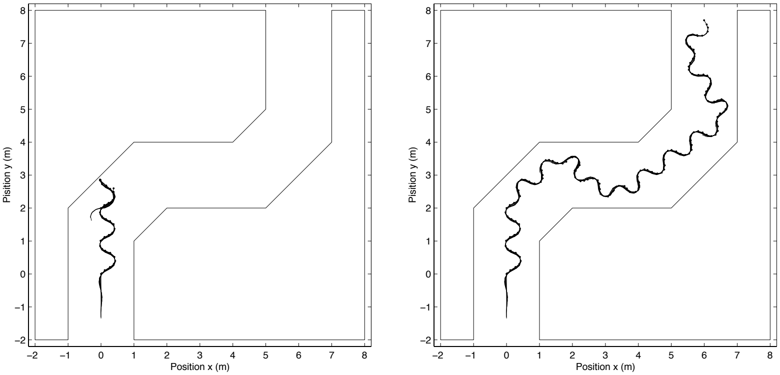
Natural Actor Critic in Snake Domain



(a) Crank course

(b) Sensor setting

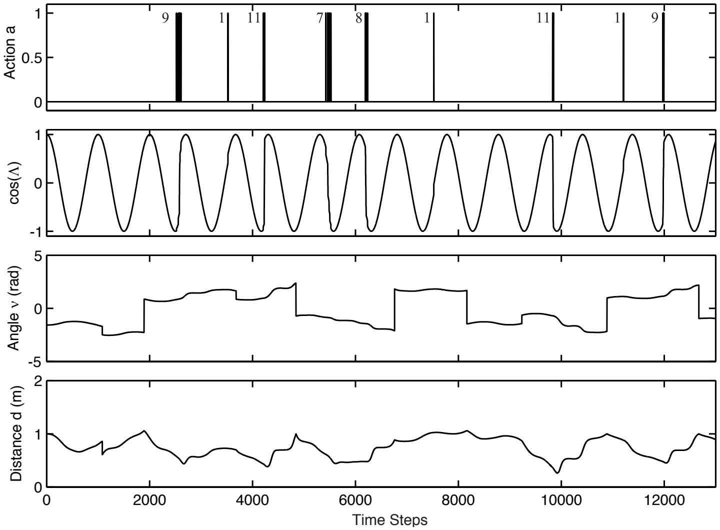
Natural Actor Critic in Snake Domain (2)



(a) Before learning

(b) After learning

Natural Actor Critic in Snake Domain (3)



Summary of Policy Gradient Algorithms

- The policy gradient has many equivalent forms

$$\begin{aligned}
\nabla_\theta J(\theta) &= \mathbb{E}_{\pi_\theta}\left[\nabla_\theta \log \pi_\theta(s,a)\; v_t\right] && \text{REINFORCE} \\
&= \mathbb{E}_{\pi_\theta}\left[\nabla_\theta \log \pi_\theta(s,a)\; Q^w(s,a)\right] && \text{Q Actor-Critic} \\
&= \mathbb{E}_{\pi_\theta}\left[\nabla_\theta \log \pi_\theta(s,a)\; A^w(s,a)\right] && \text{Advantage Actor-Critic} \\
&= \mathbb{E}_{\pi_\theta}\left[\nabla_\theta \log \pi_\theta(s,a)\; \delta\right] && \text{TD Actor-Critic} \\
&= \mathbb{E}_{\pi_\theta}\left[\nabla_\theta \log \pi_\theta(s,a)\; \delta e\right] && \text{TD}(\lambda)\text{ Actor-Critic} \\
G_\theta^{-1}\nabla_\theta J(\theta) &= w && \text{Natural Actor-Critic}
\end{aligned}$$

- Each leads a stochastic gradient ascent algorithm
- Critic uses policy evaluation (e.g. MC or TD learning) to estimate $Q^\pi(s,a)$, $A^\pi(s,a)$ or $V^\pi(s)$